Processes, fork and thread

Hajussüsteemide loeng

Parallelism

- So far the main focus has been on parallelism for a large number of machines running on the net.

- Now we will look at parallelism inside a single machine.

Main parallelism capabilities in one machine

- May have several processors (rare)
- A single processor typically contains N cores (e.g. i7-10875H has 8 cores), each one basically a separate processor.
- Intel processor has 2*N “threads”: these are not the threads we will look at the lecture, but a clever trick to run two processes *somewhat* in parallel on a single core.

A note about cache

- Typical current large processors have 3 layers of cache memory:
  - Level 1 and Level 2 are private for each core
  - Level 3 is shared by all cores (and some other subsystems)
- For example, i7-10875H has

Cache L1:	64K (per core)
Cache L2:	256K (per core)
Cache L3:	12MB (shared)

Other sources of parallelism

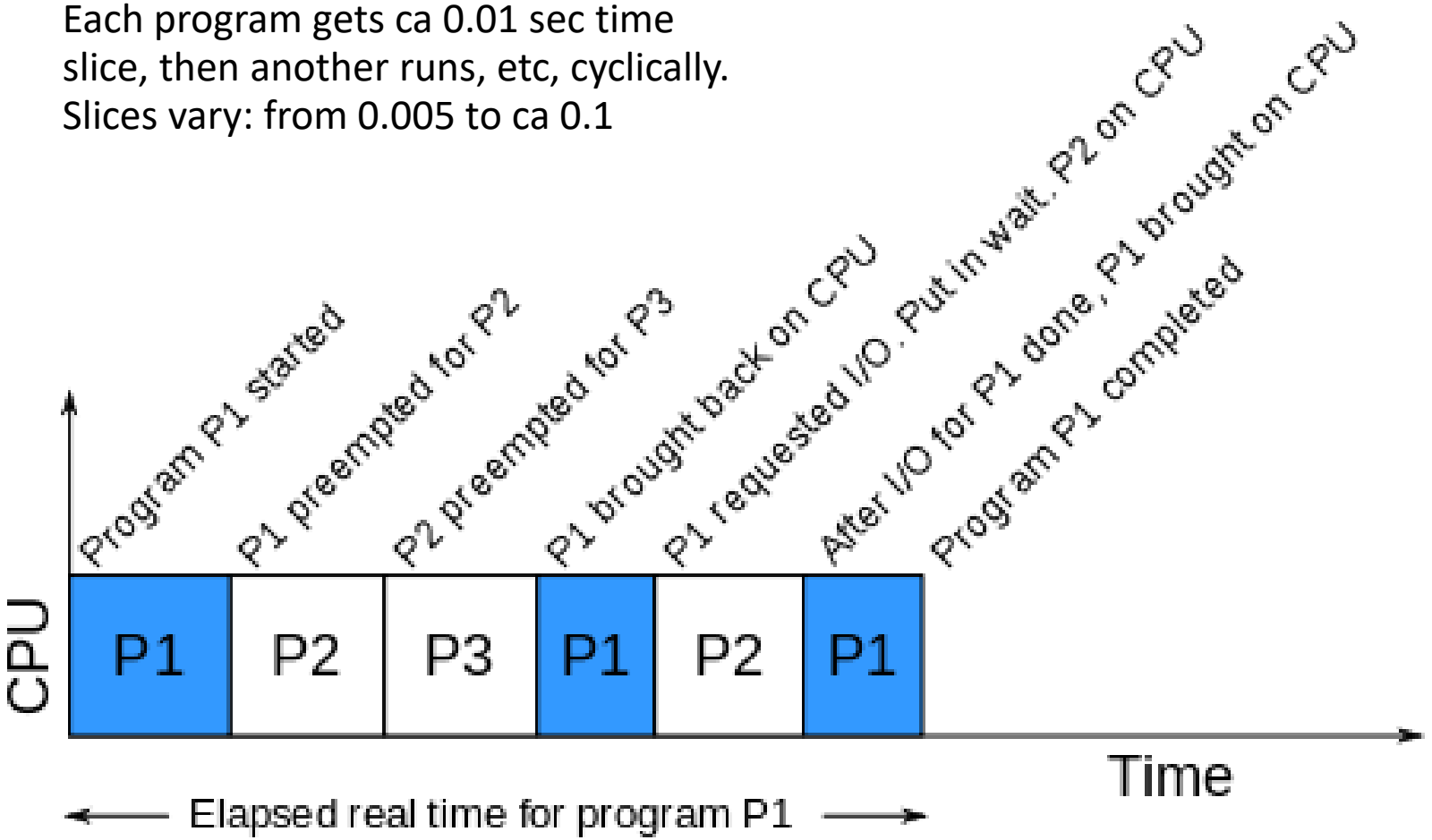
- A single process running on a single core: the core actually runs a lot of things in parallel and speculatively (predicting branches): this is managed by hardware and hard to control by a program.

- The graphics card is a highly parallel processor (thousands of small cores) with a different instruction set and different ways to control: stuff we talk about in this lecture does not apply there.

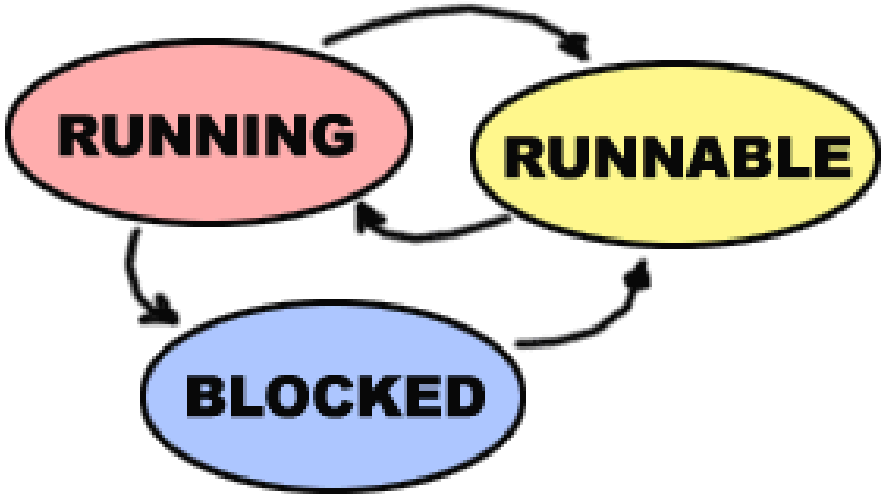
Processes

Running programs in parallel

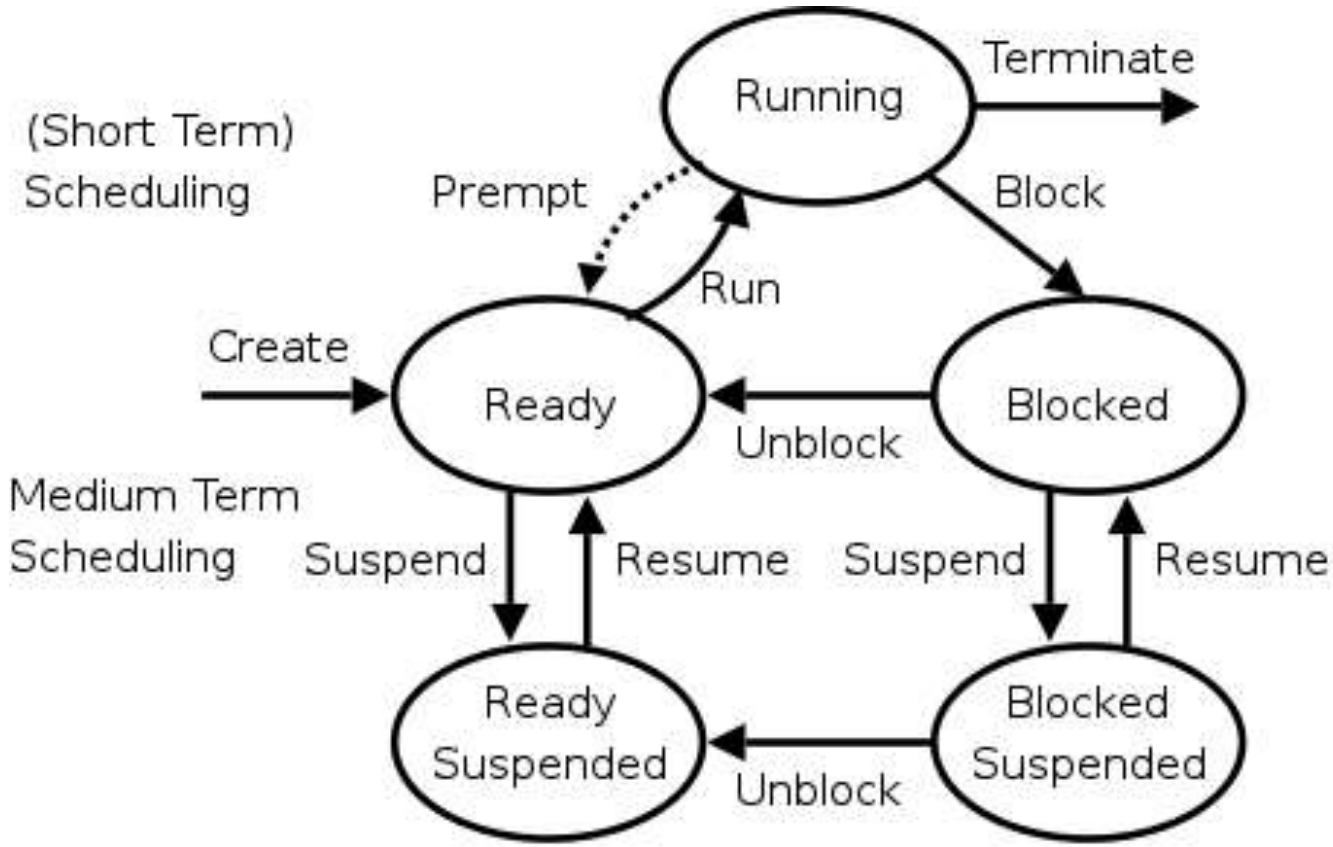
Each program gets ca 0.01 sec time slice, then another runs, etc, cyclically.
Slices vary: from 0.005 to ca 0.1



Processes wait and run

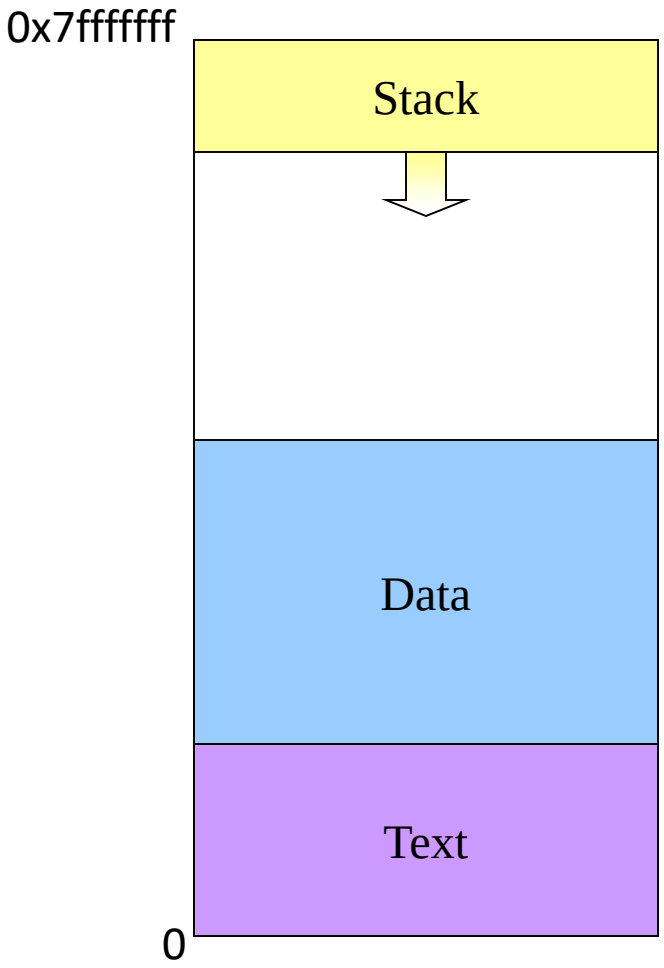


Process lifecycle in more details



Unblock is done by another task (a.k.a. wakeup, release, V)
Block is a.k.a. sleep, request, P)

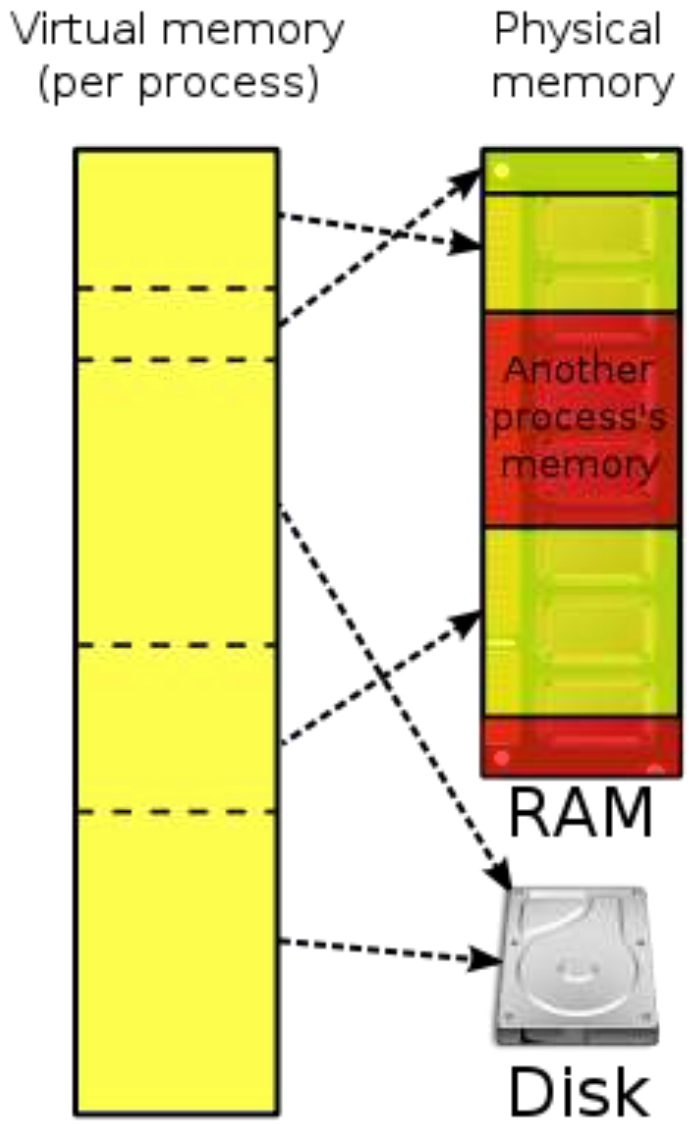
Inside a (Unix) process



- Processes have three segments
  - Text: program code
  - Data: program data
    - Statically declared variables
    - Areas allocated by malloc() or new
  - Stack
    - Automatic variables
    - Procedure call information
- Address space growth
  - Text: doesn't grow
  - Data: grows "up"
  - Stack: grows "down"

Virtual memory

- Important: OS gives a “simulated” block of consecutive memory to each process.
- Physically the parts of the “consecutive” memory may come from different areas of physical memory.
- The smallest memory “chunk” used here is called a “memory page” and is typically 4 kilobytes.

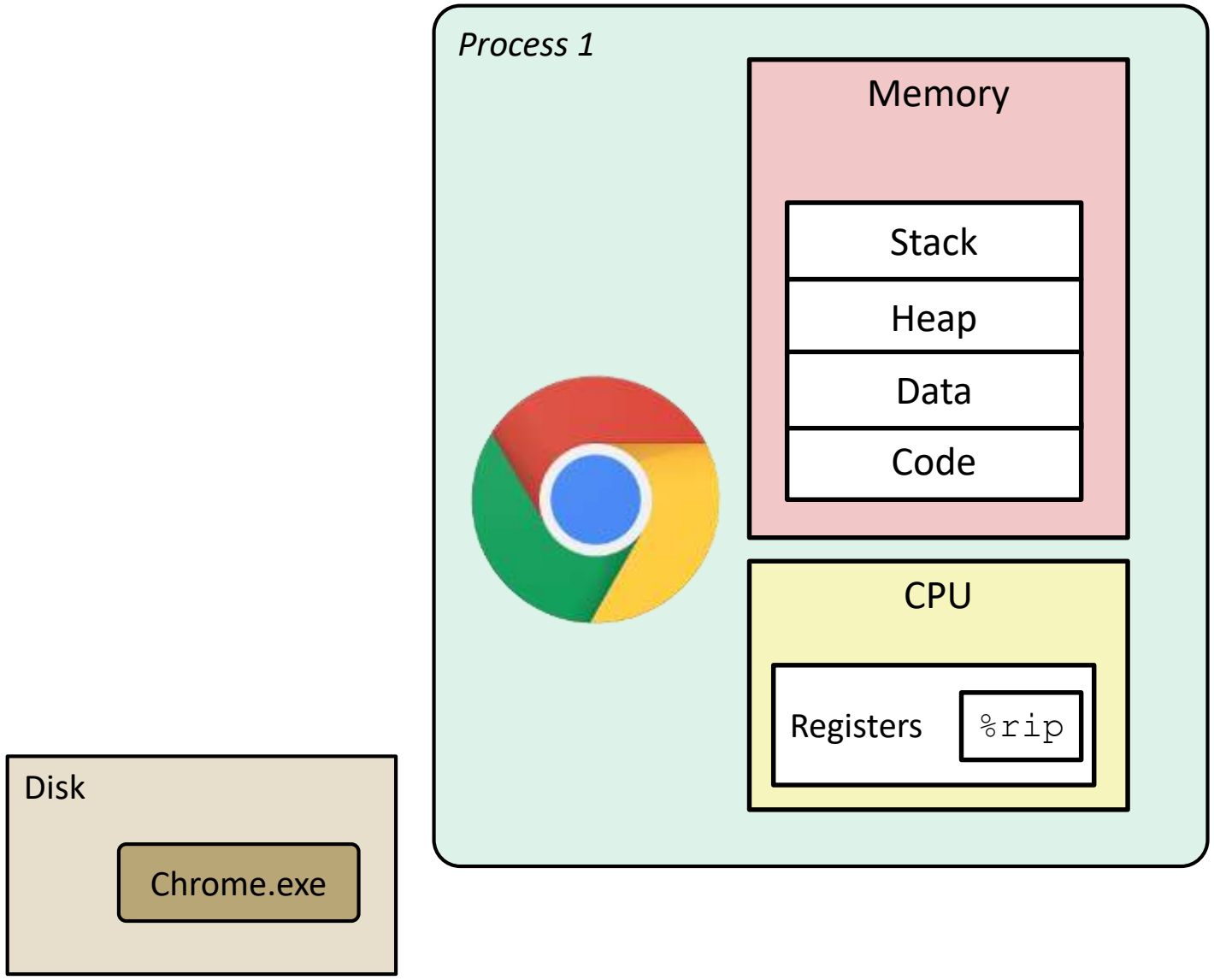


Some nice slides from

https://courses.cs.washington.edu/courses/cse351/17au/lectures/20/CSE351-L20-processes_17au.pptx

What is a process?

It's an *illusion*!



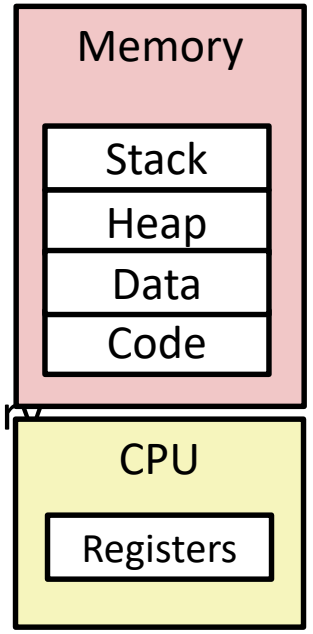
What is a process?

- Another *abstraction* in our computer system
  - Provided by the OS
  - OS uses a data structure to represent each process
  - Maintains the ***interface*** between the program and the underlying hardware (CPU + memory)
- What do *processes* have to do with *exceptional control flow*?
  - Exceptional control flow is the *mechanism* the OS uses to enable **multiple processes** to run on the same system
- What is the difference between:
  - A processor? A program? A process?

hardware the "blueprint" an instance

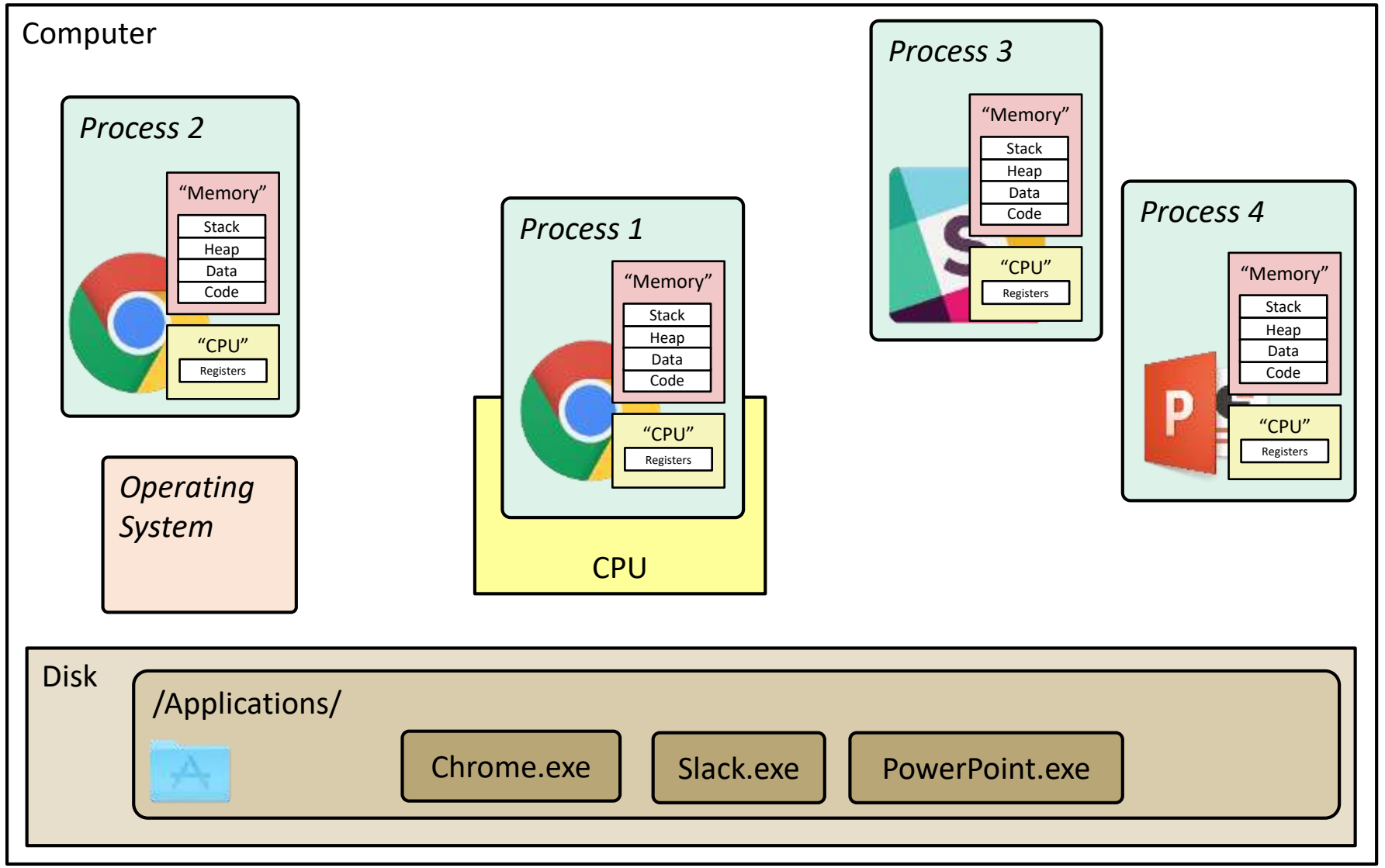
Processes

- A ***process*** is an instance of a running program
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”
- Process provides each program with two key abstractions:
  - *Logical control flow*
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called ***context switching***
  - *Private address space*
    - Each program seems to have exclusive use of main memory
    - Provided by kernel mechanism called ***virtual memory***



What is a process?

It's an *illusion*!



Making parallel processes

- Fork: make a copy of the whole process with its own separate memory. The main mechanism on UNIXes

- Thread: make a parallel process which shares memory with the parent. On UNIXes and Windows.

.

Fork

Fork: making processes in UNIX

- In UNIXES (linux, MacOs, Android, …) the main way to make new processes is the fork system call
- fork() creates a copy of the running process.
- A newly forked process has its own memory, completely separate and independent from the parent.

Fork and memory: copy-on-write

- Suppose you have a process A using 1 GB of memory.
- Now you fork() a copy getting its own separate 1GB of memory.
- Should the contents of all the initial 1GB be copied to the new 1GB?
- Answer: copy-on-write
  - No, OS will map the same physical memory pages to two different virtual memory areas / addresses, no copying of content.
  - However, when one process writes to some page in memory, then a new real copy with copied contents of this page (page: 4 kilobytes) is made before

```
#include  <stdio.h>
#include  <sys/types.h>
#define   MAX_COUNT  200
void  ChildProcess(void);               /* child process prototype  */
void  ParentProcess(void);              /* parent process prototype */

void  main(void) {
     pid_t  pid;
     pid = fork();
     if (pid == 0)   ChildProcess();
     else  ParentProcess();
}

void  ChildProcess(void) {
     int   i;
     for (i = 1; i <= MAX_COUNT; i++)
          printf("   This line is from child, value = %d\n", i);
     printf("   *** Child process is done ***\n");
}

void  ParentProcess(void){
     int   i;
     for (i = 1; i <= MAX_COUNT; i++)
          printf("This line is from parent, value = %d\n", i);
     printf("*** Parent is done ***\n");
}
```

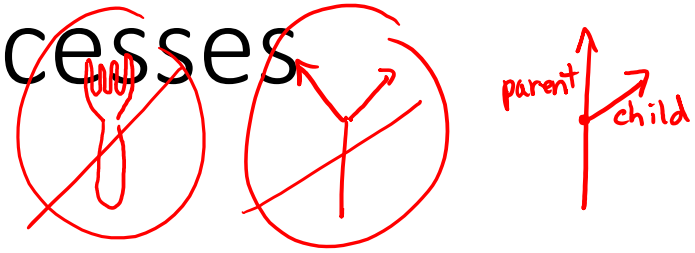
Some nice slides with more forking from

https://courses.cs.washington.edu/courses/cse351/17au/lectures/20/CSE351-L20-processes_17au.pptx

Creating New Processes & Programs

- fork-exec model (Linux):
  - `fork()` creates a copy of the current process
  - `exec*()` replaces the current process' code and address space with the code for a different program
    - Family: `execv`, `execl`, `execve`, `execle`, `execvp`, `execlp`
  - `fork()` and `execve()` are *system calls*
    - ↳ intentional, synchronous exceptions ⇒ traps
- Other system calls for process management:
  - `getpid()`
  - `exit()`
  - `wait()`, `waitpid()`

`fork`: Creating New Processes



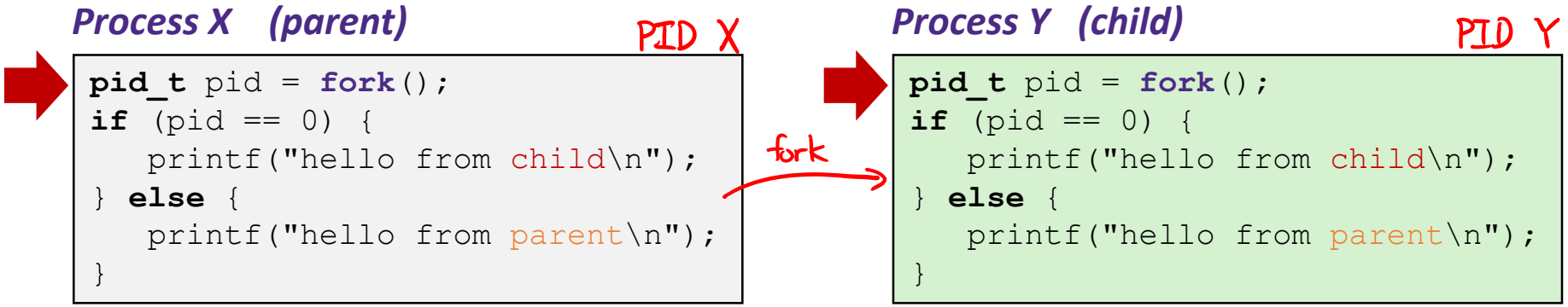
- **`pid_t`** `fork(`**`void`**`)`
  - Creates a new "child" process that is *identical* to the calling "parent" process, including all state (memory, registers, etc.)
  - Returns 0 to the child process
  - Returns child's process ID (PID) to the parent process
- Child is *almost* identical to parent:
  - Child gets an identical (but separate) copy of the parent's virtual address space
  - Child has a different PID than the parent

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

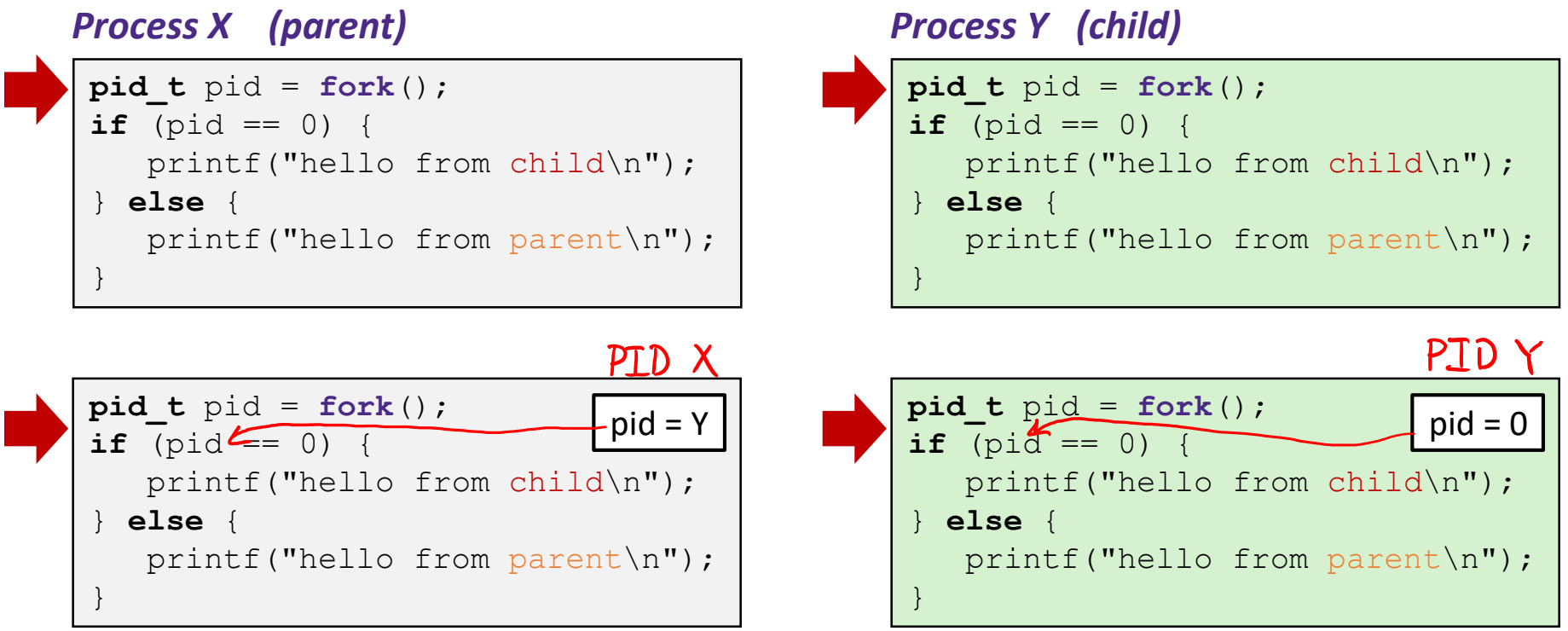
parent gets child's PID
child gets 0

- `fork` is unique (and often confusing) because it is called once but returns "twice"

Understanding fork



Understanding fork



Understanding fork

Process X (parent)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Process Y (child)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = Y

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

`hello from parent`

`hello from child`

Which one appears first?

non-deterministic!

Yet another fork Example

```
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

splits here

← child only

← parent only

← both

- Both processes continue/start execution after `fork`
  - Child starts at instruction after the call to `fork` (storing into `pid`)
- Can't predict execution order of parent and child
- Both processes start with `x=1`
  - Subsequent changes to `x` are independent
- Shared open files: stdout is the same in both parent and child

Exec: a companion to fork

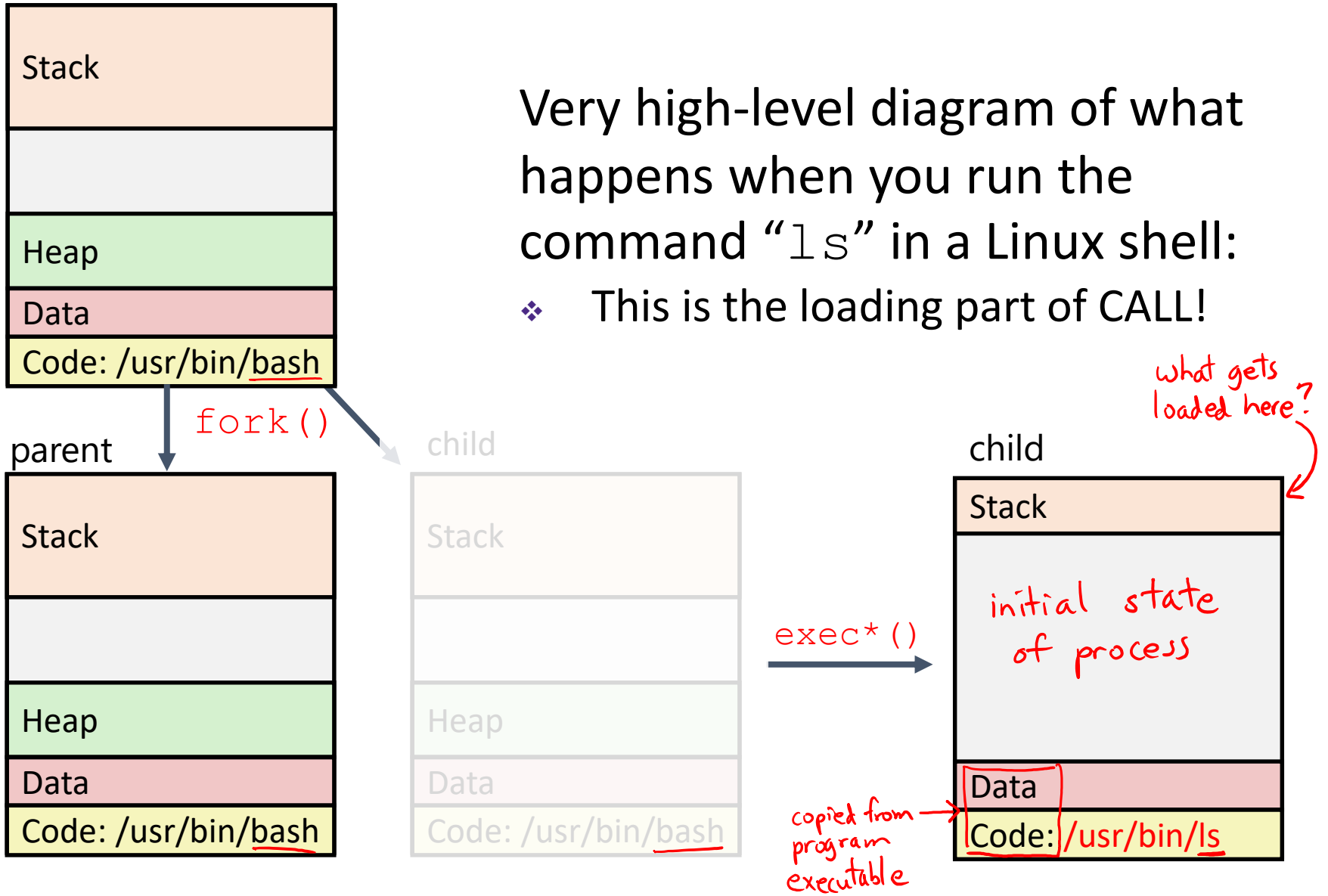
Fork-Exec

Note: the return values of `fork` and `exec*` should be checked for errors

- fork-exec model:
  - `fork()` creates a copy of the current process
  - `exec*()` replaces the current process' code and address space with the code for a different program
    - Whole family of `exec` calls – see **`exec(3)`** and **`execve(2)`**

```
// Example arguments: path="/usr/bin/ls",
//     argv[0]="/usr/bin/ls", argv[1]="-ahl", argv[2]=NULL
void fork_exec(char *path, char *argv[]) {
   pid_t pid = fork();
   if (pid != 0) {
      printf("Parent: created a child %d\n", pid);
   } else {
      printf("Child: about to exec a new program\n");
      execv(path, argv);
   }
   printf("This line printed by parent only!\n");
}
```

Exec-ing a new program



Very high-level diagram of what happens when you run the command "`ls`" in a Linux shell:

- This is the loading part of CALL!

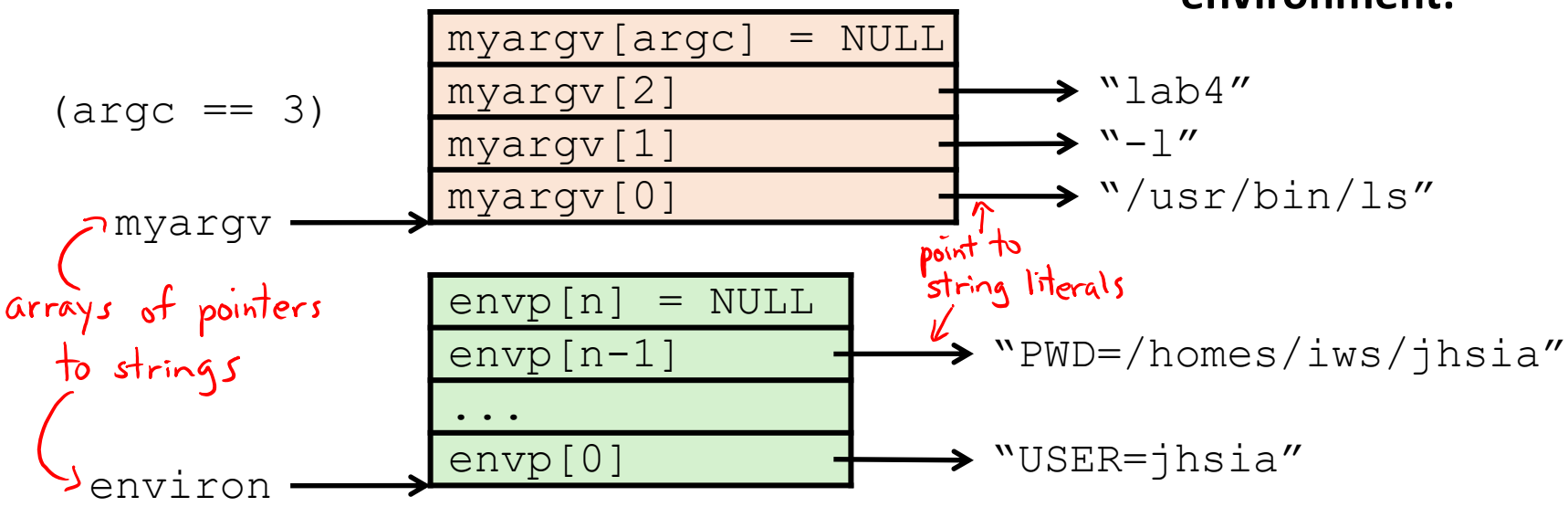
execve Example

int main(int argc, char* argv[])
get command-line arguments into program

This is extra (non-testable) material

Execute "/usr/bin/ls -l lab4" in child process using current environment:

arguments




```
if ((pid = fork()) == 0) {    /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Run the `printenv` command in a Linux shell to see your own environment variables

`exit`: Ending a process

- **`void`** `exit(`**`int`** `status)`
  - Exits a process
    - Status code: 0 is used for a normal exit, nonzero for abnormal exit

Zombies

- When a process terminates, it still consumes system resources
  - Various tables maintained by OS
  - Called a "zombie" (a living corpse, half alive and half dead)

- *Reaping* is performed by parent on terminated child
  - Parent is given exit status information and kernel then deletes zombie child process

- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (pid == 1)
    - **Note:** on more recent Linux systems, `init` has been renamed `systemd`
  - In long-running processes (e.g. shells, servers) we need *explicit* reaping

`wait`: Synchronizing with Children

- **`int`** `wait(`**`int *`**`child_status)`
  - Suspends current process (*i.e.* the parent) until one of its children terminates
  - Return value is the PID of the child process that terminated
    - *On successful return, the child process is reaped*
  - If `child_status != NULL`, then the `*child_status` value indicates why the child process terminated
    - Special macros for interpreting this status – see **`man wait(2)`**
- **Note:** If parent process has multiple children, `wait` will return when *any* of the children terminates
  - `waitpid` can be used to wait on a specific child process

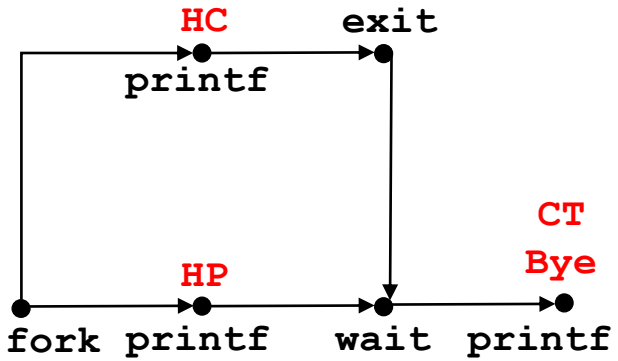
wait: Synchronizing with Children

```
void fork_wait() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
      exit(0);
   } else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
}
```

forks.c

child

parent



Feasible output:

HC HP
HP HC
CT CT
Bye Bye

Infeasible output:

HP
CT
Bye
HC

Process cooperation

with slides from

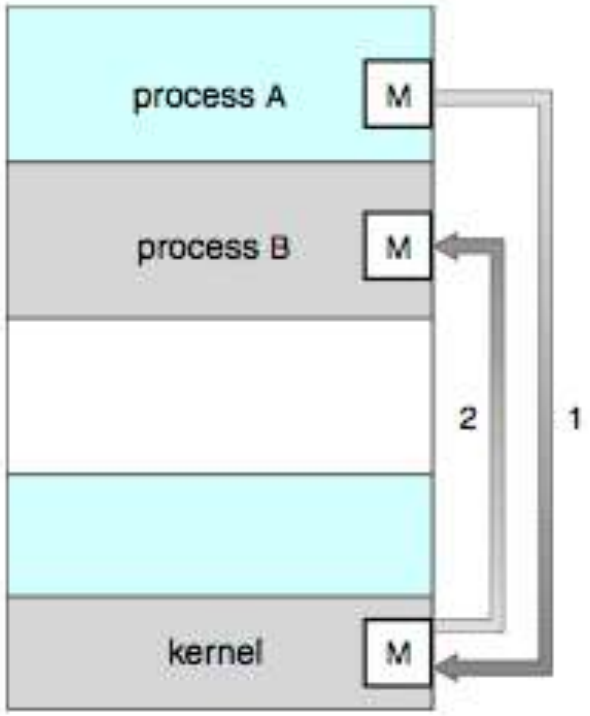
http://www.cs.columbia.edu/~junfeng/09sp-w4118/lectures/l6-proc-linux.ppt

Generic context: Cooperating Processes

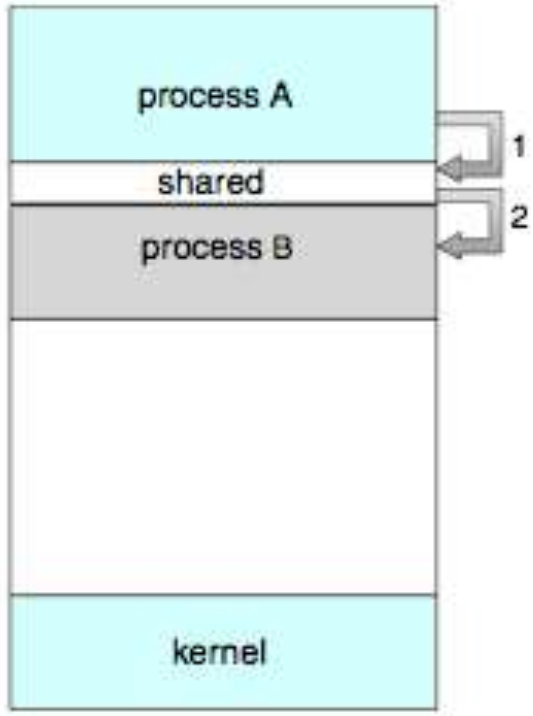
- Independent process cannot affect or be affected by the execution of another process.
- Cooperating process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity/Convenience

Interprocess Communication (IPC) Models

Message Passing



Shared Memory



Message Passing v.s. Shared Memory

- Message passing
  - Why good? Simpler. All sharing is explicit
  - Why bad? Overhead. Data copying, cross protection domains
- Shared Memory
  - Why good? Performance. Set up shared memory once, then access w/o crossing protection domains
  - Why bad? Synchronization

IPC Example: Unix signals

- Signals
  - A very short message: just a small integer
  - A fixed set of available signals. Examples:
    - 2: SIGINT, sent (usuallly) when you press ctrl+C
    - 9: SIGKILL, to kill a process
    - 11: SIGSEGV, sent when there is a memory error
- Send a signal to a process
  - kill(pid_t pid, int sig)
  - Signal can be sent by users, kernel, or other processes
- What to do when receiving a signal? Installing a handler for a signal
  - sighandler_t signal(int signum, sighandler_t handler);

IPC Example: Unix pipe

- int pipe(int fd[2]);
  - Returns two file descriptors in fd[0] and fd[1];
  - Writes to fd[1] will be read on fd[0]
  - When last copy of fd[1] closed, fd[0] will return EOF
  - Return 0 on success, -1 on error

- Operations on pipes:
  - read/write/close --- as with files
  - When fd[1] closed, read(fd[0]) returns 0 bytes
  - When fd[0] closed, write(fd[1]):
    - Kills process with SIGPIPE, or if blocked
    - Failes with EPIPE

IPC Example: Unix pipe (cont.)

```
int pipefd[2];
pipe(pipefd);
switch(pid=fork()) {
case -1: perror("fork"); exit(1);
case 0:  close(pipefd[0]);
         // write to fd[1]
         break;
default:  close(pipefd[1]);
         // read from fd[0]
        break;
}
```

IPC Example: Unix Shared Memory

- int shmget(key_t key, size_t size, int shmflg);
  - Create a shared memory segment, and return its id
  - key: unique identifier of a shared memory segment, or IPC_PRIVATE (means create a new shared mem seg)
- void* shmat(int shmid, const void *addr, int flg)
  - Attach shared memory segment to address space of the calling process. Return a pointer to shared memory
  - shmid: id returned by shmget()
- int shmdt(const void *shmaddr);
  - Detach from shared memory

IPC Example: Unix Shared Memory (cont.)

```
int id = shmget(IPC_PRIVATE, sizeof(int),
                        IPC_CREAT | 0666);
int *x = (int*)shmat(id, NULL, 0);
*x = 0;
switch(pid=fork()) {
case -1: perror("fork"); exit(1);
case 0: while(1) { ++*x; sleep(1); }
default: while(1) { printf("x = %d\n", *x); sleep(1); }
}
```

Problem: synchronization! (later)

Threads

Using several slides from Jerry Breecher

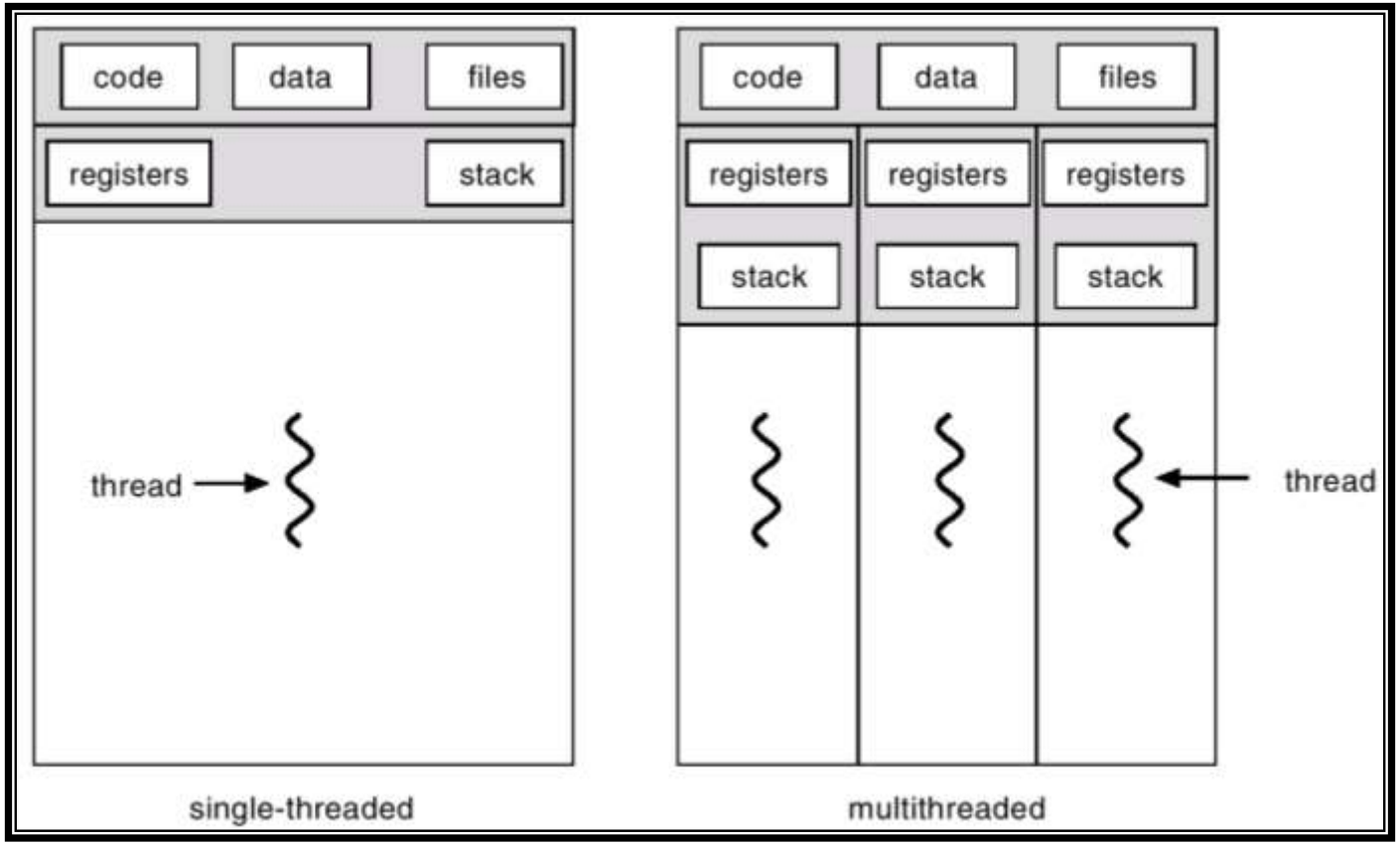
Main point

- A thread is a parallel process sharing memory with the parent
- Only the stack and registers (basically, local variables) are unique for each thread
- All the global variables and data structures are shared

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr );

main() {
pthread_t thread1, thread2;
char *message1 = "Thread 1";
char *message2 = "Thread 2";
int iret1, iret2;

/* Create independent threads each of which will execute function */

iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

/* Wait till threads are complete before main continues. Unless we */
/* wait we run the risk of executing an exit which will terminate */
/* the process and all threads before the threads have completed. */

pthread_join( thread1, NULL);
pthread_join( thread2, NULL);

printf("Thread 1 returns: %d\n",iret1);
printf("Thread 2 returns: %d\n",iret2);
exit(0);
}

void *print_message_function( void *ptr ) {
char *message;
message = (char *) ptr;
printf("%s \n", message);
}
```

THREADS

Single and Multithreaded Processes



THREADS

User Threads

- Thread management done by user-level threads library

- Examples
  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris *threads*

Kernel Threads

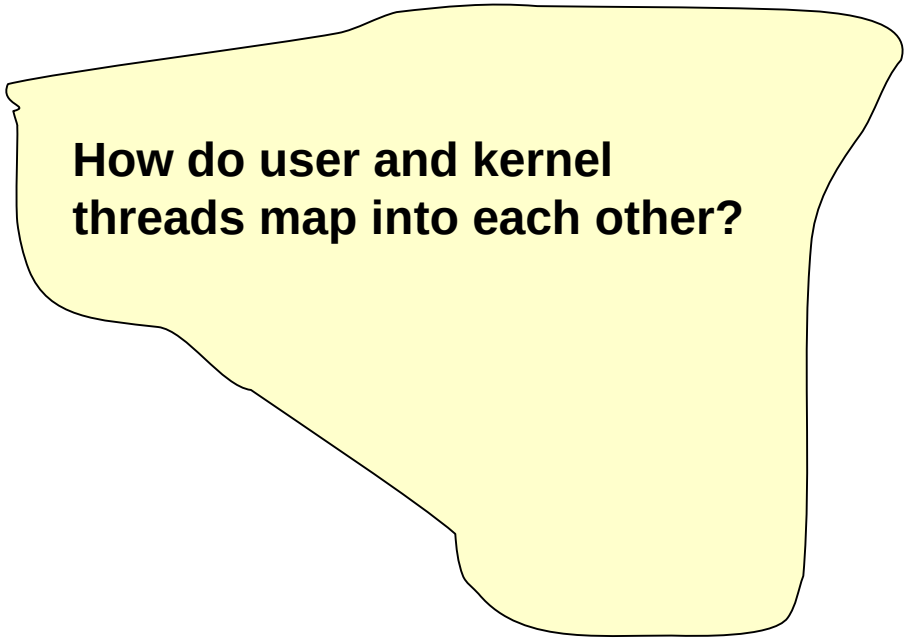
- Supported by the Kernel

- Examples
  - Windows 95/98/NT/2000
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux

THREADS

Multithreading Models

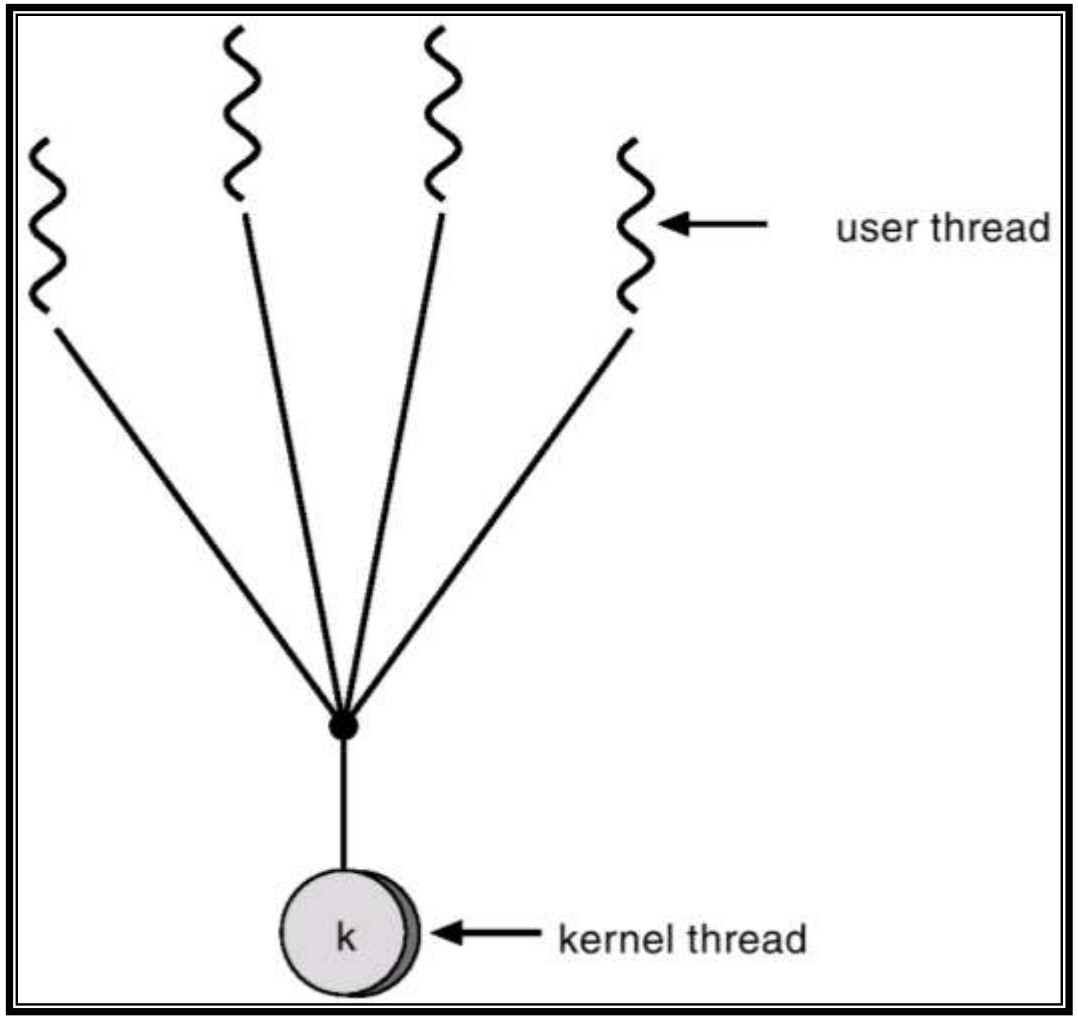
- Many-to-One
- One-to-One
- Many-to-Many



THREADS

Many-to-One

- Many user-level threads mapped to single kernel thread.
- Used on systems that do not support kernel threads.
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

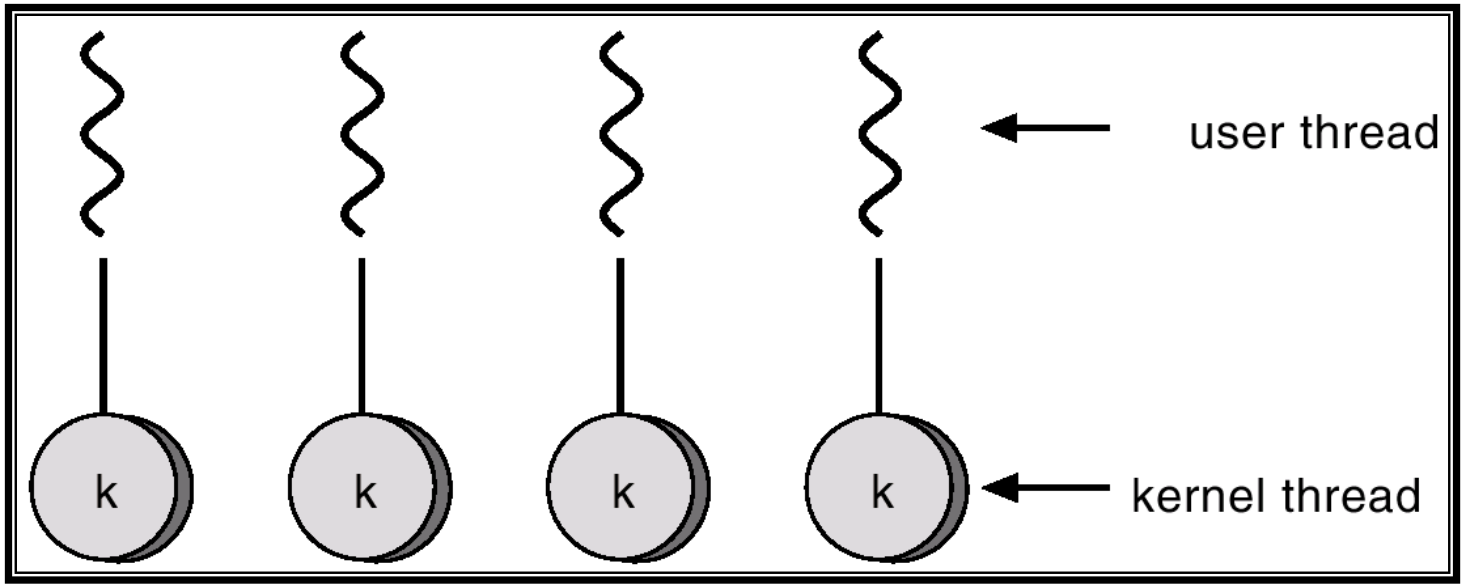


THREADS

One-to-One

- Each user-level thread maps to kernel thread.

- Examples
  - Windows 95/98/NT/2000
  - Linux



THREADS

Semantics of fork() and exec() system calls

- Does **fork()** duplicate only the calling thread or all threads?

Thread cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Threading Issues

Signal handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- Options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific threa to receive all signals for the process

Thread pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

Thread specific data

- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Scheduler activations

- Many:Many models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads

THREADS Various Implementations

PThreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Windows Threads

- Implements the one-to-one mapping
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the **context** of the threads

THREADS Various Implementations

Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

Java Threads

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- Java threads are managed by the JVM.